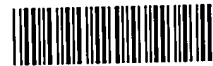# APPLICATION

FOR

UNITED STATES LETTERS PATENT

TITLE: EFFICIENT CONNECTION AND MEMORY
MANAGEMENT FOR MESSAGE PASSING ON A
SINGLE SMP OR A CLUSTER OF SMPS

APPLICANTS: Narayan VENKATSUBRAMANIAN,
Terry D. DONTJE and
Steve SISTARE

22511

PATENT TRADEMARK OFFICE

EFFICIENT CONNECTION AND MEMORY MANAGEMENT FOR MESSAGE PASSING ON A SINGLE SMP OR A CLUSTER OF SMPS

Background of Invention

Field of the Invention

[0001]   The present invention generally relates to message passing between multiple processes and more particularly to message passing in a multiple computer processor system.

Background Art

[0002]   A computer system executes applications and programs in order to perform necessary and desired functions. Typically, one computer processor is responsible for executing the applications of the computer system in which the processor resides. However, there is a trend toward using multiple processors for executing applications.

[0003]   Such multiprocessor systems have a plurality of processors, in close communication with one another, sharing system resources, i.e., memory, peripheral devices, etc. Thus, by using a multiprocessor system, execution throughput can be increased.

[0004]   A typical architecture used by multiprocessor systems is the symmetric multiprocessing ("SMP") architecture, in which each of a number of processors in a machine share the memory available on the machine. SMP architecture provides fast performance by making different processors available to a number of processes on the machine simultaneously. In order to gain further performance out of SMP systems, many such systems are connected together and the processes

running on each system communicate with each other via the use of some inter-process communication paradigm. A group of SMPs connected together to run an application is called a cluster of SMPs, and each SMP system in the cluster is referred to as an SMP node.

[0005]   In a cluster of SMPs, different SMP nodes may have different amounts of processes. In order for the cluster of SMP nodes to execute and run a computer application, i.e., a user program, processes running on the processors in the same and/or different SMP nodes communicate with each other.

[0006]   Several mechanisms have been developed to facilitate transfer of data among processors, and more specifically, between processing node memories. One mechanism for exchanging messages is referred to as "message passing." Several known message passing specifications have been developed, including Message Passing Interface ("MPI") and Parallel Virtual Machine ("PVM"). Generally, in message passing, in order to transfer data from one process to another, the transferring process generates a message including the data and transfers the message to another process. For example, when a first process needs to retrieve data from a second process, the first process generates a message, including a data retrieval request, and transfers the message to the second process from which data is to be retrieved. Subsequently, the second process executes the data retrieval request and transfers the data to the first process in a message as described above.

[0007]   In a computer system constructed according to a distributed memory architecture, messages using the message passing mechanism are transferred between processing nodes over a network and processed or otherwise handled by a respective processing node when they arrive at a respective destination. In a computer system constructed according to a shared memory architecture, several buffer arrangements have been developed to facilitate message transfer.

2

[0008]   Figure 1 shows a typical prior art embodiment of a shared memory architecture message transfer mechanism in which a plurality of processes are provided with a plurality of buffers, each associated with a particular process pair. Figure 1 shows a first SMP node (20), on which two processes, an 'i-th' process (24) and a 'k-th' process (26), reside, and a second SMP node (22), on which a 'j-th' process (36) resides. When the 'i-th' process (24) transfers a message to the 'j-th' process (36), the 'i-th' process (24) deposits the message into a buffer, B(i,j) (38), that is specifically maintained to store data for the 'i-th' process (24) and the 'j-th' process (36) when the 'i-th' process (24) is a sending process, i.e., a process that sends a message, and the 'j-th' process (36) is a receiving process, i.e., a process that receives a message. Similarly, when the 'j-th' process (36) transfers a message to the 'i-th' process (24), the 'j-th' process (36) deposits the message into a buffer, B(j,i) (28), that is specifically maintained to store data for the 'j-th' process (36) and the 'i-th' process (24) when the 'j-th' process (36) is the sending process and the 'i-th' process (24) is a receiving process.

[0009]   Once a message is deposited into a particular buffer, the receiving process can copy the contents of that buffer to its region of memory. For example, once the 'i-th' process (24) has deposited a message to B(i,j) (38), the 'j-th' process (36) can then retrieve the message by copying it to its region of memory, after which the 'i-th' process (24) can again transfer a message to the 'j-th' process (36).

[0010]   The mechanism described above for message transfers between the 'i-th' process (24) and the 'j-th' process (36) can also be applied to message transfers between the 'i-th' process (24) and the 'k-th' process (26), using buffers B(i,k) (30) and B(k,i) (32), and between the 'j-th' process (36) and the 'k-th' process (26), using buffers B(k,j) (40) and B(j,k) (34).

[0011]   Allocating buffers in the manner described above with reference to Figure 1 ensures that there is no contention for buffers, e.g., memory space, between

3

processes attempting to transmit messages to the same process. However, since each buffer has to be sufficiently large to be able to accommodate a large message, it is appreciated that a significant portion of memory address space may be required to maintain the plurality of buffers. Further, the portion of memory address space needed to maintain a plurality of buffers increases as the number of processes increase.

[0012]   In another mechanism, shown in Figure 2, shared buffers are provided that include a plurality of buffers available to all processes. Figure 2 shows a first SMP node (also referred to as "SMP Node 1") (42), on which an 'i-th' process (50), a 'k-th' process (52), and a first shared buffer pool (54), reside, and a second SMP node (also referred to as "SMP Node 2") (44), on which a 'j-th' process (62) and a second shared buffer pool (66) reside. Typically, buffers in the first and second shared buffer pools (54, 66) have sizes on the order of the sizes of the process pair buffers (28, 30, 32, 34, 38, 40) shown in Figure 1. In addition to being provided a shared buffer pool, a process is provided with a relatively small buffer (also referred to as "postbox," "pbx," or "postbox portion") to transfer messages to a particular process.

[0013]   When the 'i-th' process (50) transfers a message to the 'j-th' process (62), the 'i-th' process (50) first attempts to store the message in a postbox, pbx(i,j) (60), that is specifically maintained for message transfers between the 'i-th' process (50) and the 'j-th' process (62) when the 'i-th' process (50) is a sending process and the 'j-th' process (62) is a receiving process. If the message fits in pbx(i,j) (60), then the message is stored there for subsequent retrieval by the 'j-th' process (62). However, if the message does not fit in pbx(i,j) (60), the 'i-th' process (50) allocates a buffer from the 'j-th' process's (62) shared buffer pool (66), i.e., the second shared buffer pool (66), loads the message into the allocated buffer, and loads a pointer to the allocated buffer in pbx(i,j) (60) for the 'j-th' process (62). Thereafter, the 'j-th' process (62) can detect that a message or

4

pointer has been loaded into pbx(i,j) (60) and then retrieve the message or pointer. If pbx(i,j) (60) contains a pointer, the 'j-th' process (62) can use the pointer to identify the allocated buffer from its shared buffer pool (66) and then retrieve the message by copying it to its region of memory. After the 'j-th' process (62) has retrieved the message from the allocated buffer from its shared buffer pool (66), the allocated buffer, i.e., memory, can then be returned to its shared buffer pool (66). In the case that the 'j-th' process (62) transfers a message to the 'i-th' process (50), the 'j-th' process (62) loads a message or pointer into a postbox, pbx(j,i) (46), that is specifically maintained for message transfers between the 'i-th' process (50) and the 'j-th' process (62) when the 'j-th' process (62) is the sending process and the 'i-th' process (50) is the receiving process.

[0014]   The mechanism described above with reference to Figure 2 for message transfers between the 'i-th' process (50) and the 'j-th' process (62) can also be applied to message transfers between the 'i-th' process (50) and the 'k-th' process (52), using postboxes pbx(k,i) (48) and pbx(i,k) (56), and between the 'j-th' process (62) and the 'k-th' process (52), using postboxes pbx(k,j) (64) and pbx(j,k) (58).

[0015]   The message transfer mechanism described with reference to Figure 2 reduces the amount of memory space that is needed relative to the amount of memory space that is needed in the message transfer mechanism described in Figure 1, because the postboxes (46, 48, 56, 58, 60, 64) shown in Figure 2 require less space than the buffers (28, 30, 32, 34, 38, 40) shown in Figure 1. Moreover, the amount of shared buffer pools increases linearly with the amount of processes. However, some contention for memory space is possible with the message transfer mechanism described in Figure 2. Typically, a lock/unlock mechanism is provided to synchronize access to shared buffer pools to ensure that a current sending process does not deposit a message into a particular buffer in the shared buffer pool in the case that a receiving process has not retrieved a message from a

5

prior sending process that used the same buffer in the shared buffer pool that the current sending process is attempting to send a message to. However, lock/unlock mechanisms, which are used to stop subsequent processes from altering memory before a receiving process can copy that memory, can become a performance bottleneck when a large number of processes transfer messages simultaneously.

[0016]   Figure 3 shows another prior art mechanism used to handle message transfer. This mechanism uses a sender owned buffer that is managed by each sending process. Figure 3 depicts nodes 1...n (70, 72, 74, 76), where n represents the number of nodes in a particular system. In order to depict the mechanism shown in Figure 3 more clearly, processes on nodes 2...n (72, 74, 76) transfer messages to processes on node 1 (70). However, those skilled in the art will appreciate that processes on node 1 (70) can transfer messages to processes on nodes 2...n (72, 74, 76).

[0017]   Nodes 1...n (70, 72, 74, 76) each have processes 1...m, where m represents the number of processes on a particular node. Note that m can be different for different nodes. Each of the processes on nodes 1...n (70, 72, 74, 76) is allocated a region of common memory, which it uses in its processing operations. In the case of multiple nodes as shown in Figure 3, each process is allocated a piece of memory on each node including the node on which the process itself resides. With reference to Figure 3, the allocated memory regions (78, 80, 82) for processes 1 and 2 on node 2 (72) and process m on node n (76) are shown, respectively. Other processes shown on nodes 1...n (70, 72, 74, 76) also have allocated memory regions which are not shown in Figure 3.

[0018]   The allocated memory regions (78, 80, 82) are each divided into p postbox blocks and a buffer pool having q buffer blocks, where p represents the number of processes on the node to which it is sending a message and where q represents the number of buffer blocks in a particular buffer pool.

[0019]   If process 1 on node 2 (72) needs to send a message to process 2 on node 1 (70), process 1 on node 2 (72) first attempts to load the message into a postbox in its allocated memory region (78) that is specifically maintained for transfers between process 1 on node 2 (72) and process 2 on node 1 (70) when process 1 on node 2 (72) is the sending process and process 2 on node 1 (70) is the receiving process. If the message is successfully loaded into the postbox, then process 2 on node 1 (70) can thereafter retrieve the message by copying it from the postbox to its region of common memory.

[0020]   However, if the message is not small enough to fit in the postbox, then process 1 on node 2 (72) selects one of the buffer blocks from its allocated memory region (78), loads the message into that selected buffer, and stores a pointer to the selected buffer into the postbox that is specifically maintained for transfers between process 1 on node 2 (72) and process 2 on node 1 (70) when process 1 on node 2 (72) is the sending process and process 2 on node 1 (70) is the receiving process. Thereafter, process 2 on node 1 (70), the receiving process, can retrieve the message by first retrieving the pointer from the postbox described above, and then using the pointer stored in the postbox by process 1 on node 2 (72), the sending process, to identify the buffer block into which the message was loaded, and then copying the message from the buffer block into its common region of memory. Afterwards, the receiving process notifies the sending process that it can reuse the postbox for a future message.

[0021]   Note that the mechanism described in Figure 3 can also be applied for message transfers between any of the process shown in Figure 3.

[0022]   Moreover, with reference to Figure 3, the number of allocated memory regions that need to be maintained by a system is known ahead of time since each sending process owns one allocated memory region. Also, since each allocated memory region is owned and maintained by one process, a lock/unlock

mechanism is not needed.

Summary of Invention

[0023]   The present invention provides new and improved ways to allocate memory for message passing to facilitate message transfer in a single shared memory symmetric multiprocessor ("SMP") or a cluster of SMPs.

[0024]   In one aspect, a method of transmitting a first message from a first process on a first node to a second process on a second node comprises the first process defining a first process memory space on the second node, the first process assigning a separate postbox portion of the first process memory space for each respective process on the second node, and the first process storing the first message in the first process memory space and a first message indicator in the postbox portion of the first process memory space assigned to the second process.

[0025]   In another aspect, a method of transmitting a first message from a first process to a second process on a second node comprises the first process determining if a postbox portion for the second process can store a first message indicator for the first message, the first process determining if there is sufficient unallocated first process memory space available to store the first message if the postbox portion for the second process can store the first indicator for the first message, the first process storing the first message in the first process memory space if there is sufficient unallocated first process memory space, and storing the first message indicator in the postbox portion of the first process memory space assigned to the second process.

[0026]   In another aspect, a computing node comprises a first process memory space including a separate postbox portion for each process on the node, and a means for each process on the computing node to access a respective postbox portion to retrieve a message indicator identifying a location in the first process

8

memory space of a message from the first process to the respective process on the computing node.

[0027]   In another aspect, a method for managing memory when a process on a first node attempts to store a current message in a memory space on a second node comprises determining whether any acknowledgment signals have been received by the first node from a receiver of a previously sent message that used memory in the memory space, selectively reclaiming a postbox associated with the previously sent message if an acknowledgment signal has been received from the receiver of the previously sent message, and storing the current message if the postbox associated with the previously sent message is reclaimed.

[0028]   In another aspect, a method for handling message transfers by selectively transferring a message in pieces comprises determining whether a size of a remaining message for a receiving process is greater than a cyclic transaction size, and storing the remaining message into a memory space if the size of the remaining message for the receiving process is not greater than the cyclic transaction size.

[0029]   In another aspect, a method for recovering memory used for message transfers comprises maintaining a receiver list to track a plurality of receiving processes, wherein the plurality of receiving processes on the receiver list are maintained in an order in which the plurality of receiving process were communicated to by a sending process.

[0030]   In another aspect, a method for recovering memory used for message transfers comprises attempting to recover memory used by a plurality of receiving processes by traversing a receiver list, and selectively recovering memory from those receiving processes on the receiver list that reside on a node which is the same as the node on which a receiving process resides.

[0031]   In another aspect, a method for creating and handling a data memory

segment and an acknowledgment segment when a first process on a first node needs to send a message to a second process on a second node comprises determining whether there is a data memory segment for the first process on the second node, creating the data memory segment for the first process on the second node if there is not already a data memory segment for the first process on the second node, determining whether there is an acknowledgment segment on the first node, and creating the acknowledgment segment on the first node if there is not already an acknowledgment segment on the first node.

[0032]   Other aspects and advantages of the invention will be apparent from the following description and the appended claims.

Brief Description of Drawings

[0033]   Figure 1 shows a prior art embodiment of a message passing mechanism that uses a buffer for every process pair.

[0034]   Figure 2 shows another prior art embodiment of a message passing mechanism that uses postboxes and shared buffer pools.

[0035]   Figure 3 shows another prior art embodiment of a message passing mechanism that uses buffers owned by sending processes.

[0036]   Figure 4 shows a block diagram in accordance with an embodiment of the present invention.

[0037]   Figure 5 shows a block diagram in accordance with an embodiment of the present invention.

[0038]   Figure 6 shows a flow process in accordance with an embodiment of the present invention.

[0039]   Figure 7 shows a flow process in accordance with the embodiment shown in Figure 6.

[0040]   Figure 8 shows a flow process in accordance with the embodiment shown in Figure 7.

Detailed Description

[0041]   The present invention relates to a method for improving memory management for message transfer between processes on symmetric multiprocessor ("SMP") nodes.   The invention also relates to a method for establishing connections between processes on SMP nodes. Further, the invention relates to a method for improving SMP process communication using a remote shared memory ("RSM") protocol. The RSM module allows an area of memory on one computer to be visible or accessible as part of another computer's memory, i.e., address space.

[0042]   Figure 4 shows a block diagram of a message transfer mechanism in accordance with an exemplary embodiment of the present invention. Figure 4 shows a cluster of two SMP nodes, SMP Node 1 (90) and SMP Node n (92), along with memory segments, i.e., memory space, used for message passing. Although Figure 4 shows a cluster of only two SMP nodes, the following discussion with reference to Figure 4 can be applied to any number of SMP nodes.

[0043]   SMP Node 1 (90) and SMP Node n (92) are connected by a network "**R**" (shown in Figure 4), which, for the purposes of this embodiment, supports the RSM protocol. SMP Node 1 (90) contains processes P(1,1) through P(1,m), where m represents the number of processes on SMP node 1 (90). SMP Node n (92) contains processes P(n,1) through P(n,m'), where m' represents the number of processes on SMP node n (92). Note that the first ordinal number for a process indicates the number of the node on which the process resides.

[0044]   Prior to the commencement of any message transfer, the processes have to create memory segments on respective nodes before depositing messages for

11

processes on those nodes. The creation of these segments may occur on a per segment basis when required. The postponement of segment creation until segment creation is required is called lazy segment creation, and is described below with reference to Figure 6, Figure 7, and Figure 8.

[0045]   Each process P(n,m) is allocated memory segments G(m,1) through G(m,n) on its node and other nodes in a cluster, where m represents the process number and n represents the number of the node on which the process or allocated memory segment resides. For example, a process P(1,2) can communicate with a process P(3,4) by depositing a message in memory segment G(2,3), which resides on node 3.

[0046]   Note that the second ordinal number for a process indicates the process number on only the node indicated by the first ordinal number. In other words, the second ordinal is only bounded by the number of processes on the node indicated by the first ordinal. For example, the process P(1,2) refers to the second process on the first node whereas process P(3,4) refers to the fourth process on the third node. In other words, the second ordinal is only bounded by the number of processes on the node indicated by the first ordinal.

[0047]   In addition to delaying memory segment creation until it is necessary, simplex, i.e., one way, connections for message passing may be established. Such connections may be used when a process sends a message to another process in a cluster without message transfer in the reverse direction. In this situation, the process that received the message establishes a memory segment (96) (also referred to in Figure 4 as "**ACK_SEG**") to write an acknowledgment indication (referred to as "ACK") on the node on which the process that sent the message resides.

[0048]   The creation of ACK memory segments, like the one (96) shown in Figure 4, may also be done in a lazy manner such that an ACK memory segment is

dynamically created on a first node to receive ACKs from processes on a second node when a process from the first node sends a message to a process on the second node. The structure of the ACK memory segment (96) is described in detail with reference to Figure 5.

[0049]   Referring still to Figure 4, process P(1,1) on SMP node 1 (90) communicates with process P(n,1) by first attempting to deposit a message in a postbox in memory segment G(1,n) (94) on SMP node 2 (92). If the message is too large to fit within a postbox in G(1,n) (94), then the message is loaded into a buffer block allocated from a buffer pool in G(1,n), and the sending process, P(1,1) stores a pointer to that buffer block in the postbox. Thereafter, the receiving process, P(n,1) may retrieve the message by copying the message from the postbox to its region of memory, or in the latter case, by using the pointer stored in the postbox by process P(1,1) to identify the buffer block into which the message was loaded, and then copying the message from the buffer block to its common region of memory. Process (n,1) then acknowledges the receipt of the message by writing an ACK message into a corresponding ACK slot in **ACK_SEG** (96).

[0050]   When a sending process sends a message, the percentage of memory that can be used for the message is dependent upon the amount of memory available in the sending process's buffer pool and the number of receivers that the sending process may potentially send messages to. This ensures that a sending process cannot use all the memory in its buffer pool for sending a message to one receiving process. The value of the percentage of memory that can be used by a sending process for a message may be pre-defined by a system user or dynamically modified by the system during actual operation.

[0051]   Although the discussion with reference to Figure 4 involves process P(1,1) sending a message to process P(n,1), the same procedure may be applied to all

processes on SMP node 1 (90) and SMP node n (92) (as shown in Figure 4). Further, Figure 4 shows message passing and ACK writing between various sets of processes aside from between P(1,1) and P(n,1).

[0052]   In an alternative embodiment, the creation of a sending process's buffer pool on a remote node, i.e., a node on which the receiving process resides that is different from the node on which the sending process resides, is delayed until the sending process needs to send a message to a receiving process on that remote node.

[0053]   In another embodiment, memory allocation is reduced by establishing simplex connections between process pairs. Simplex connections, as opposed to duplex connections, are one way communication channels. In this embodiment, a sending process can send a message to a receiving process on a remote node and receive acknowledgments from the receiving process indicating to the sending process that the message was received without the establishment of a data buffer. For example, a process P(i) can send data to process P(j) on a remote node, but P(j) never needs to send data back to P(i). The acknowledgment indication is still feasible (as described below) and net memory connection is reduced because duplex connection establishments are delayed until necessary.

[0054]   Referring to Figure 5, a structure of an acknowledgment ("ACK") memory segment (also referred to as "ACK segment") (96) is shown in accordance with an exemplary embodiment of the present invention. As discussed above, during the process of memory segment creation used for depositing messages, an ACK memory segment is dynamically created on a first node to receive ACKs from processes on a second node when a process from the first node sends a message to a process on the second node.

[0055]   Consider that the ACK segment (96) shown in Figure 5 is created for SMP Node 1 and SMP Node 2 and resides on SMP Node 1. If SMP Node 1 has m

processes and SMP Node 2 has m' processes, then the ACK memory segment (96) is first divided into m sections, one section for each process on SMP Node 1, and each section, in turn, is further divided into m' ACK blocks, one ACK block for each process on SMP Node 2. Each ACK block consists of k slots, one for each postbox used for sending messages between processes on both the nodes in a node pair (a pair of SMP nodes).

[0056]   When a process P(i) on SMP Node 1 sends a message to process P(j) on SMP Node 2 (not shown) by depositing a message into a memory segment G(i,2) (according to the mechanism described above with reference to Figure 4), P(j) acknowledges the receipt of the message by writing an ACK message, i.e., an acknowledgment indication, into the corresponding ACK slot location in an ACK block "j" in section "i" of the ACK segment on SMP Node 1.

[0057]   The creation of ACK segments is done in a "lazy" manner such that ACK segments are created on a node to receive ACKs if a process from that node sent a message to a process on the node for which the ACK segment is being created for. This technique is time efficient, because it saves time by not creating all potential ACK segments at the start of a system run. Further, this technique is space efficient, because it avoids creating memory segments that may not be required at all.

[0058]   Figure 6 shows an exemplary flow process in accordance with an embodiment of the present invention. The process begins when a first process, P1, on a first node, N1, has a message to send to a second process, P2, on a second node, N2 (step 100). Thereafter, the flow process entails determining whether P1 has a memory segment, G1, on N2 (step 102). If P1 does not have a G1 memory segment on N2, a G1 memory segment on N2 is created (step 104). After creating the G1 memory segment on N2 (step 104) or if there already was a G1 memory segment on N2, the flow process next entails determining whether there is an ACK

segment, A1, on N1 (step 106). If there is not an ACK segment on N1, an ACK segment on N1 is created (step 108). After creating the ACK segment on N1 (step 108) or if there already was an ACK segment on N1, the flow process enters a memory recovery stage (109).

[0059]   The memory recovery stage (109) entails determining whether there is space in the G1 memory segment on N2 for the message (step 110). If there is not space for the message in the G1 memory segment on N2, memory in the G1 memory segment is recovered (step 125) until enough space has been recovered to allow the message to be stored in the G1 memory segment. Once enough memory is recovered to store the message in the G1 memory segment or if there already was enough memory to store the message in the G1 memory segment, the flow process exits the memory recovery stage (109), and P1 on N1 sends a message to P2 on N2 (step 150).

[0060]   In another embodiment of the present invention, a message sent by a sending process to a receiving process is broken down into smaller pieces such that the memory used to send an earlier piece of the message can be reused to send a later piece of the message to the receiving process.

[0061]   In an alternative embodiment of the present invention, a sending process maintains a circular list of receiving processes to which it has sent messages. The receiving processes are maintained on the list in the order they were communicated to by a sending process, e.g., the receiving process to which a message was sent last is at the end of the list. The list is circular because the last receiving process on the list is followed by the first receiving process on the list. In other embodiments, the receiving processes may be maintained on the list according to the order which receiving processes received pieces of one or more messages.

[0062]   In another embodiment of the present invention that implements the

circular list mentioned above, a count of the number of buffer pool blocks used is maintained for each receiving process. If a sending process needs to allocate buffer pool blocks, i.e., memory, to a receiving process and has no memory available, the sending process attempts to recover memory allocated to the same receiving process earlier and reuses the memory if such memory is available. Receiving processes that do not consume any buffer pool blocks are removed from the circular list. In other embodiments of the present invention, the sending process can traverse its circular list and recover buffer pool blocks, i.e., memory, it used earlier to send to other receiving processes on the same node as the node on which the receiving process that the sending process is trying to send to resides. Further, these embodiments of the present invention may be used in a single SMP system or in a multiple SMP environment, e.g., RSM model.

[0063]   Figure 7 shows an exemplary flow process in accordance the embodiment shown in Figure 6. Specifically, Figure 7 shows a detailed flow process within the memory recovery stage (109). The flow process shown in Figure 7 is an expansion of the steps of the memory recovery stage (109) shown in Figure 6. Further, the embodiment in Figure 7 uses circular lists as described above.

[0064]   The flow process begins with entering the memory recovery stage (109) and then determining whether a current message will fit into a postbox (step 112). If the current message will fit into a postbox, the flow process next determines whether a postbox is available (step 114). If a postbox is not available, an attempt is made to reclaim other postboxes and buffers (step 116). The flow process continues to reclaim postboxes and buffers (step 116) until a postbox becomes available (step 114). Once a postbox does become available, the current message is deposited into the now available postbox and the flow process exits the memory recovery stage (109).

[0065]   In the case that a message will not fit into a postbox, a determination is

17

made as to whether the size of a remaining message, i.e., a message sent before the current message that is still waiting to be fully sent, is greater than a maximum cyclic transaction size at which a message can be sent (step 120). If the remaining message's size is greater than the cyclic transaction size, the remaining message is broken into smaller pieces such that the size of potential message transfers is less than or equal to the cyclic transaction size (step 122). Once a remaining message is broken down into smaller pieces (step 122) or if the size of the remaining message was already less than or equal to the cyclic transaction size, the flow process next entails determining whether any buffer blocks are available (step 124) to store the remaining message or piece thereof.

[0066]   If there are any buffer blocks available, the remaining message or piece thereof are stored into an available buffer block (step 126). Next, a pointer is set up and written in a postbox to indicate the location of the buffer block in which the remaining message or piece thereof was stored (steps 128 and 130). Thereafter, the flow process entails determining whether a receiver, i.e., a receiving process to which a message is ultimately sent, for the remaining message is on a receiver list, i.e., a list that is maintained to track receiving processes (step 132). If the receiver is not on the receiver list, the receiver is added to the receiver list (step 134). After adding the receiver to the receiver list (step 134) or if the receiver was already on the receiver list, a buffer count for the receiver, i.e., an amount of buffers that the receiver is using, is incremented (step 136). Thereafter, the remaining message or piece thereof is sent and the remaining size of the remaining message is decremented by the amount sent (step 138). Next, the flow process entails determining whether there are any remaining messages left (step 139). If there are not any remaining messages left, including the current message, the flow process exits the memory recovery stage (109). However, if there are remaining messages left, the flow process cycles back to the step that entails determining whether the size of the next remaining message is greater than the cyclic transaction size (step

18

120).

[0067]   In the case the no buffer blocks are available, an attempt is made to reclaim postboxes and buffers previously used for the receiver of the remaining message (step 140). Next, the flow process entails determining whether any buffer blocks have become available (step 142) after the attempt to reclaim postboxes and buffers for the receiver of the remaining message (step 140). If any buffer blocks become available, the flow process stores the remaining message or piece thereof into an available buffer block (step 126) and the flow process continues therefrom as described above.   However, if no buffer blocks become available after the attempt to reclaim postboxes and buffers for the receiver of the remaining message (step 140), the flow process next entails determining whether there are other processes on the receiver list (step 144).   If there are no other processes on the receiver list, the flow process goes to back to determining whether any buffer blocks are available (step 124) and continues therefrom as described above. However, if there are other processes on the receiver list, the receiver at the head of the receiver list is picked (step 145), and an attempt is made to reclaim postboxes and buffers from the picked receiver (step 146).   The flow process next entails determining whether any buffer blocks have become available (step 147). If any buffer blocks have become available, the flow process stores the remaining message or piece thereof into an available buffer block (step 126) and the flow process continues therefrom as described above.   However, if no buffer blocks become available, the flow process next entails determining whether there are any other receivers on the receiver list (step 148).   If there are other receivers on the receiver list, the flow process picks a next receiver (step 149), attempts to reclaim buffers and postboxes for that next receiver (step 146), and continues therefrom as described above.   However, if there are no other receivers on the receiver list, the flow process goes to back to determining whether any buffer blocks are available (step 124) and continues therefrom as described above.

19

[0068]   In other embodiments, the exemplary flow process shown in Figure 7 may be applied to a plurality of process pairs on the same or different nodes transferring messages in parallel.

[0069]   Referring to Figure 8, an exemplary flow process is shown in accordance with the embodiment shown in Figure 7. Specifically, Figure 8 shows the flow process (also referred to as "memory reclamation stage (160)") corresponding to the reclaiming of postboxes and buffers as is done in steps 116, 140, and 146 of Figure 7.

[0070]   First, the flow process entails determining whether any ACKs from earlier sends, i.e., messages sent, have been received (step 162). If no ACKs have been received from earlier sends, this indicates that no receiving process has received a message since the last time the flow process checked whether any ACKs from earlier sends had been received, and the flow process consequently exits the memory reclamation stage (160).

[0071]   However, if any ACKs have been received from earlier sends, this indicates that memory has become available because a receiving process has received a message and relinquished memory associated with storing the message after the receiving process copied it into its region of memory. In this case, the flow process attempts to reclaim postboxes associated with the ACKs received from earlier sends (step 164). The flow process next entails determining whether a reclaimed postbox contains a pointer to a buffer block (step 166). If there is not a pointer to a buffer block, the flow process cycles back to checking whether any ACKs from earlier sends have been received (step 162) and continues therefrom. However, if there is a pointer to a buffer block in a reclaimed postbox, the buffer block pointed to by the pointer is designated 'free' for use by processes attempting to transfer messages (step 168). Next, the count of buffer blocks used by the receiver associated with the reclaimed buffer block is decremented by the amount

20

of buffer blocks 'freed' (step 170) (a postbox may contain more than one pointer pointing to more than one buffer block). The flow process then entails determining whether the count of buffer blocks used by that receiver is now zero (step 172). If the count of buffer blocks is zero, then that receiver is removed from the receiver list (step 174) because it no longer is associated with any unavailable memory. Thereafter, or if the buffer block count for that receiver is not zero, the flow process exits the memory reclamation stage (160).

[0072]   Advantages of the present invention may include one or more of the following. As is described above, a sending process maintains a list of receiving process to which it has sent messages. Further, receiving processes that are no longer using any memory units are deleted from the list. By using the list, the sending process can reclaim memory without having to search all receiving processes on a node. The sending process, therefore, searches among receiving processes that will or already have abandoned memory as opposed to searching through all receiving processes, which may or may not yield available memory. Further, a process can reclaim memory it used to send a message to a process and reuse it to send messages to the same process or another process on the same node. Maintaining a list of receiving processes in the manner described above reduces the time necessary to reclaim memory and makes process communication more efficient.

[0073]   Another advantage is that total memory consumption is reduced by improving the connection establishment between processes such that connections are established dynamically during the lifetime of a particular program. By delaying an establishment of a connection until a sending process needs to send a message to a remote node, the creation of a sending process's buffer pool on a remote node is accordingly delayed. This technique reduces total memory because only a subset of all potential connections is established.

[0074]   An additional advantage is that as only a subset of potential connections is established, program execution initialization time is reduced because unnecessary connections are not established during the initialization time.

[0075]   A further advantage is that memory allocation is reduced by establishing simplex connections where possible. Such simplex connections allow for a sending process to send a message to a receiving process and also allow for a sending process to receive acknowledgment indications from a receiving process without creating a reverse connection, i.e., a duplex connection. It follows that memory consumption is approximately halved and duplex connection establishment is postponed until a receiving process needs to send data aside from an acknowledgement indication back to the receiving process.

[0076]   Another advantage is that a separate acknowledgement segment is created for receiving acknowledgement indications from receiving processes. Because data and acknowledgement segments are separate buffer pools, simplex connections between a sending process and a receiving process can be maintained.

[0077]   Yet another advantage is that memory consumption is reduced because a receiving process writes an acknowledgement bit into an acknowledgement indication segment instead of having to send a message to a data segment, i.e., buffer, which can be significantly larger than a byte. This eliminates the need to create buffer pools on a node unless a process needs to write data to that node.

[0078]   While the invention has been described with respect to a limited number of embodiments, those skilled in the art, having benefit of this disclosure, will appreciate that other embodiments can be devised which do not depart from the scope of the invention as disclosed herein. Accordingly, the scope of the invention should be limited only by the attached claims.